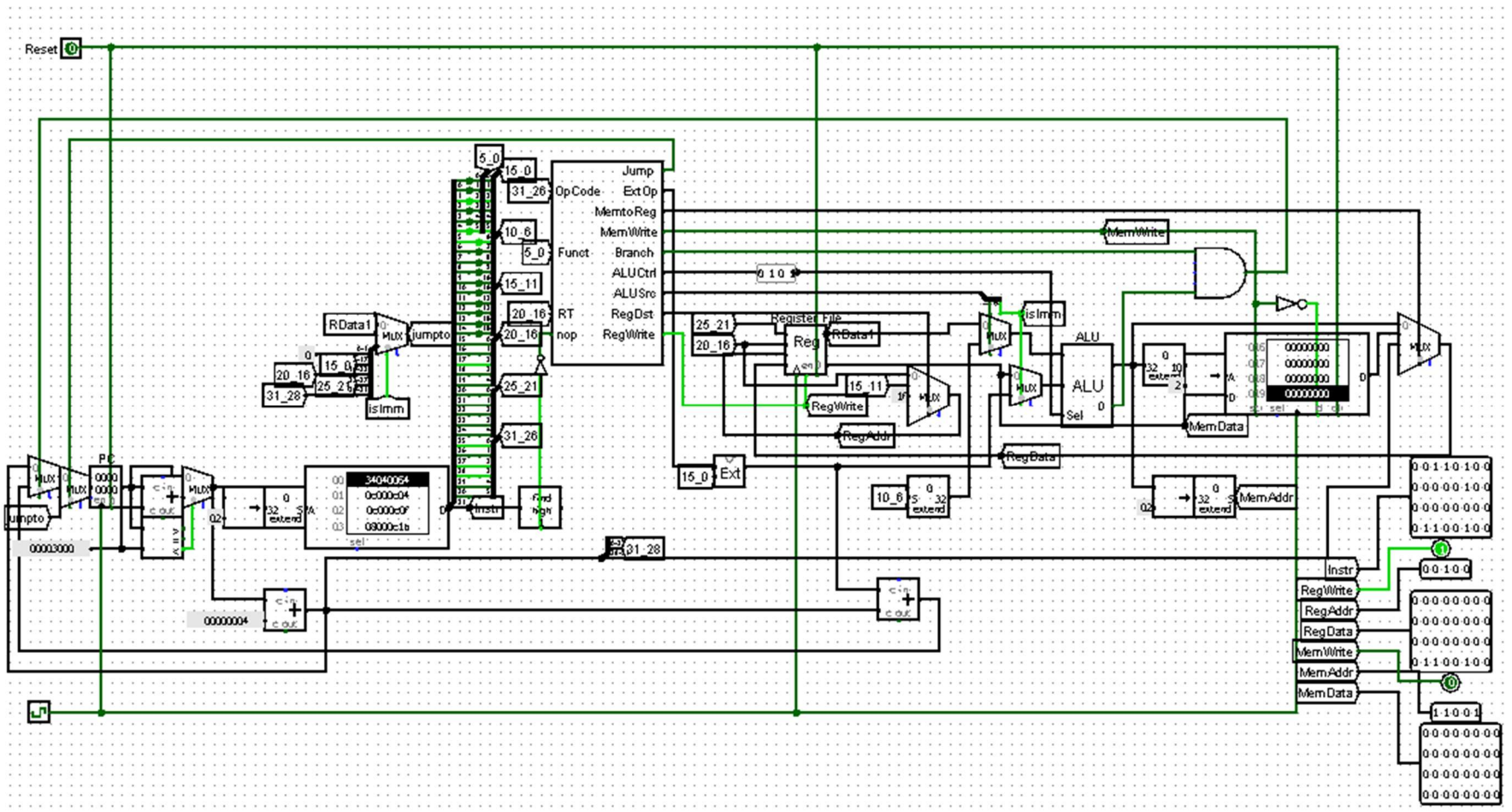
Logisim 单周期

顶层设计图



一、主要模块

1、IFU（取指令单元）。包括 PC（程序计数器，5 位），IM（指令存储器，32 位*32 字）。在主模块中分开实现，不单独建立子电路。

为了和 Mars 汇编器保持一致，PC 需要初始化为 0x30000000，因此，采取如下方法：

当 reset 信号为 1 时初始化为 0，加上 0x3000 作为 IM 的地址。之后，如果 PC 值大于等于 0x3000 则直接作为地址，否则加上 0x3000。可以证明，这个有限状态机中的值永远不小于 0x3000，但保证地址不重复加 0x3000。

表 0 IFU 端口表

端口	方向	描述
Instr[31:0]	O	取出的指令
reset	I	异步复位信号。为 1 时指令地址保持 0。

2、ALU（算术逻辑单元，4 位选择 16 种运算）

表 1 ALU 端口表

端口	方向	描述
op1[31:0]	I	操作数 1
op2[31:0]	I	操作数 2
ALUCtrl[3:0]	I	功能选择，见下表
result[31:0]	O	计算结果
zero	O	计算结果是否为 0 的指示信号

0000---非负　　0001---负数　　0010---加法　　0011---减法

0100---按位与　　0101---按位或　　0110---按位异或　0111---按位或非

1000---逻辑右移　1001---算术右移　1010---左移　　1011---相等

1100---有符号小于　1101---无符号小于　1110---正数　　1111---≤0

3、GRF（寄存器文件，32 位*32）。其中 0 号寄存器始终为 0。

表 2 寄存器文件端口

信号名	方向	描述
clk	I	时钟信号
reset	I	复位信号，将 32 个寄存器中的值全部清零 1：复位 0：无效
WE	I	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
A1	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD1
A2	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中存储的数据读出到 RD2
A3	I	5 位地址输入信号，指定 32 个寄存器中的一个作为写入的目标寄存器
WD	I	32 位数据输入信号
RD1	O	输出 A1 指定的寄存器中的 32 位数据
RD2	O	输出 A2 指定的寄存器中的 32 位数据

表 3 寄存器文件功能

序号	功能名称	描述
1	复位	reset信号有效时，所有寄存器存储的数值清零，其行为与logisim自带部件register的reset接口完全相同
2	读数据	读出 A1,A2 地址对应寄存器中所存储的数据到 RD1,RD2
3	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入 A3 所对应的寄存器中。

4、DM（数据存储器，32 位*1024 字）。用内置 RAM 实现，采用 Separate load and store ports 属性。地址 10 位。题目中要求输出 5 位地址，取后 5 位。RAM 自带时钟信号、写使能、地址端和数据端，与 DM 要求完全相同。

5、Ext（位扩展器，16→32 位，4 选择）

表 4 位扩展器端口和功能

端口	方向	描述
Imm[15:0]	I	待扩展的 16 位数
ExtOp[1:0]	I	扩展方法选择 00：符号扩展 01：无符号扩展 10：加载至高 16 位(lui) 11：符号扩展之后，左移两位
Out[31:0]	O	扩展后的 32 位数

6、Controller（指令译码器）

表 5 指令译码器端口

端口	方向	描述
Opcode[5:0]	I	指令操作码
Funct[5:0]	I	指令功能码
Jump	O	跳转信号
ExtOp[1:0]	O	位扩展方式，见表 3
MemtoReg	O	读内存信号
MemWrite	O	内存写使能信号
Branch	O	分支信号
ALUCtrl[3:0]	O	ALU 控制信号，见 ALU 模块
ALUSrc	O	ALU 操作数 2 的来源

		0：寄存器 1：立即数
RegDst	O	寄存器写地址选择 0：Instr[20:16] 1：Instr[15:11]
RegWrite	O	寄存器写使能信号

一位控制信号采用与或逻辑实现，多位控制信号采用 MUX 复用器。

二、指令解读

1、addu 指令

功能：加寄存器，不考虑溢出。`reg3=reg1+reg2`。

指令码：`000000 reg1[25:21] reg2[20:16] reg3[15:11] 100001`

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	xx	0	0	0	0010	0	1	1

2、subu 指令

功能：减寄存器，不考虑溢出。`reg3=reg1-reg2`

指令码：`000000 reg1[25:21] reg2[20:16] reg3[15:11] 100011`

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	xx	0	0	0	0011	0	1	1

3、ori 指令

功能：或立即数。`reg2=reg1 | imm`。位运算需要进行无符号扩展。

指令码：`001101 reg1[25:21] reg2[20:16] imm[15:0]`

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	01	0	0	0	0101	1	0	1

4、lw 指令

功能：按字读内存，一次读 4 个字节，放到寄存器中。`reg1= mem[reg2+imm]`。reg2 叫基地址，imm 叫做偏移量。由于偏移量可能为负，因此要进行符号扩展。寻址进行加法运算。

指令码：`100011 reg1[25:21] reg2[20:16] imm[15:0]`

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	00	1	0	0	0010	1	0	1

5、sw 指令

功能：按字写内存，一次从寄存器中向内存写 4 个字节。`mem[reg2+imm]=reg1`。reg2 叫基地址，imm 叫做偏移量。由于偏移量可能为负，因此要进行符号扩展。寻址进行加法运算。

指令码：`101011 reg1[25:21] reg2[20:16] imm[15:0]`

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite

0	00	0	1	0	0010	1	x	0

6、beq 指令

功能：当两个待比较寄存器相等时，转到分支指定的指令。

`if(reg1==reg2) goto pcnext;`

指令码：`000100 reg1[25:21] reg2[20:16] imm[15:0]`

解释：当两个寄存器相等时，`pc=pc+4+imm<<2`。否则 `pc=pc+4`。由于 imm 表示分支偏移的指令数，可以为负，因此进行符号扩展，并且左移两位。相等可以用相减等于零来判断，ALU 做减法（做异或运算也可，比减法更快）。

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	11	0	0	1	0011（或 0110）	0	x	0

7、lui 指令

功能：立即数加载至寄存器高 16 位。

指令码：`001111 00000 reg[20:16] imm[15:0]`

解释：imm 加载至寄存器 reg[31:16]。中间 5 个 0，可以认为是 0 号寄存器，因此可以选择 ext 高位扩展后与 0 号寄存器相加，ALU 做加法。

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
0	10	0	0	0	0010	0	0	1

8、nop 空指令

指令码：`0x00000000`

不执行任何操作，所有控制信号均为 0（ALUCtrl 为无关值）。

空指令不写入真值表。

9、j 指令

功能：无条件跳转至指令（常用于循环语句）

指令码：000010 imm[25:0]

解释：跳转到 imm 指定的指令地址。

控制信号：

Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
1	00	x	0	x	xxxx	x	x	0

三、数据通路

PC：输入端接 next 信号，输出端接指令存储器。

IM：输出端接指令译码器。其中 31:26 接 Opcode，5:0 接 funct。15:0 接立即数扩展 ext。25:21，20:16，15:11 接寄存器文件地址。

DM：写数据端接寄存器文件输出，读数据端接寄存器文件写数据。写地址端接 ALU 结果。MemWrite 接写使能端。

ALU：操作数接寄存器和立即数，输出端接寄存器写数据和内存写地址。还有一个零端，与 Branch 信号接在与门上，当二者均为真时分支有效。

next 信号（下一个 PC 值）：当 j 为真时，为立即数。否则当 branch 和 zero 均为真时，为 branch 计算所得地址；二者有一个为假时，为 pc+4（在 Logisim 中不允许跨地址存放数据，因此 32 位算作一个地址，地址应该加 1，但是 PC 为保持和 Mars

一致，设计成字对齐，即 4 的倍数，因此应该加 4。取地址时右移两位（或直接取[6:2]））。

选择信号均用 MUX 实现。详见开头部分的顶层图。

特别注意 next_instr 的逻辑。

四、控制器设计

首先考虑主译码器，生成一位控制信号，用与或逻辑实现。其中，addu 和 subu 的 opcode 字段均为 0，要加一个 funct 字段逻辑。每一个指令信号都是一个一位数据，用于识别是哪条指令。当某个信号为 1 时，表明当前是这条指令。必须保证两个信号不能同时为 1。

下面给出对应关系的简化真值表：

指令	opcode	funct	指令	opcode
addu	000000	100001	sw	101011
subu	000000	100011	beq	000100
ori	001101	X	lui	001111
lw	100011	X	j	000010

然后，把每个字段分成 6 个一位数据，按照上述对应关系搭建与或逻辑电路。例如：

```
j=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&~opcode[0]
```

完成后，下面需要生成控制信号了。我们把上面的控制信号表合并一下：

指令	Opcode	Funct	Jump	ExtOp	MemtoReg	MemWrite	Branch	ALUCtrl	ALUSrc	RegDst	RegWrite
addu	000000	100001	0	xx	0	0	0	0010	0	1	1
subu	000000	100011	0	xx	0	0	0	0011	0	1	1
ori	001101	xxxxxx	0	01	0	0	0	0101	1	0	1
lw	100011		0	00	1	0	0	0010	1	0	1
sw	101011		0	00	0	1	0	0010	1	x	0
beq	000100		0	11	0	0	1	0011	0	x	0
lui	001111		0	10	0	0	0	0010	0	0	1
j	000010		1	00	x	0	x	xxxx	x	x	0

然后，每一个控制信号都是一个由指令信号生成的逻辑真值。按照纵向来看，需要把每一个控制信号的真值归并起来。得到控制信号的逻辑表达式：

```
jump=j
ExtOp[0]=ori||beq ExtOp[1]=lui||beq
MemtoReg=lw
MemWrite=sw
Branch=beq
ALUSrc=ori||lw||sw
RegDst=addu||subu
RegWrite=addu||subu||ori||lw||lui
```

然后可以用或门连接控制信号和指令信号。

下面考虑 ALU 译码器。指令种类并不多，仍然可以用与或逻辑。下面给出 ALU 控制端的简化表：

指令	addu	subu	ori	lw	sw	beq	lui	j
ALUCtrl	0010	0011	0101	0010	0010	0011	0010	xxxx

得到逻辑表达式：

```
ALUCtrl[3]=0
ALUCtrl[2]=ori
ALUCtrl[1]=addu||subu||lw||sw||beq||lui
ALUCtrl[0]=subu||ori||beq
```

至此，所有译码器设计的关键已经完成。

我们看到，nop 指令根本不用放进真值表中。

五、测试

搭建完成后，任务还没有完，我们要亲手验证一下我们的处理器是否正确。先用 Mars 编一个程序段：

```
.data
arr:.space 44

.text
ori $t0,$0,0
ori $t1,$0,1
ori $s0,$0,1
ori $s1,$0,4
ori $s2,$0,40
for:
beq $t0,$s2,next
sw $t1,arr($t0)
addu $t1,$t1,$s0
addu $t0,$t0,$s1
j for
next:

ori $t0,$0,0
ori $t1,$0,0
ori $s0,$0,0
for1:
beq $t0,$s2,next1
lw $t1,arr($t0)
addu $s0,$s0,$t1
addu $t0,$t0,$s1
j for1
next1:
sw $s0,arr($t0)
li $t0,0x44897253
```

功能：先把 1-10 分别写入 arr[0:9]，然后读出它们依次累加，结果存入 arr[10]中。并专门加载一个 32 位立即数（可分解为先加载高位 lui 再或低位 ori）指令验证是否正确。

导出的机器码为

```
34080000 34090001 34100001 34110004
34120028 11120004 ad090000 01304821
01114021 08000c05 34080000 34090000
34100000 11120004 8d090000 02098021
01114021 08000c0d ad100000 3c014489
34287253
```

前面加上 v2.0 raw。打开 ROM 编辑器，导入，保存，reset。期望结果：

寄存器：$t0=0x44897253 $s0=0x37

内存：0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xa 0x37

运行，直到显示当前指令为 0x00000000。停止运行（走过几个空指令后及时停止，不要让地址溢出回到 0，因此时钟频率要调慢一点）。右击 RAM 和 GRF 查看，内存显示正确，GRF 为 8 号 0x44897253，16 号 0x00000037。由于汇编器使用了 1 号寄存器作为 lui 的结果，1 号寄存器显示为 0x44890000。说明，空指令没有执行操作，对结果没有影响。

结果正确才表明我们的设计完成。

六、思考题

1、若 PC 为 30 位，分析其与 32 位 PC 的优劣。

答：当使用 32 位 PC 时，在 MIPS 编写指令时，认为指令在内存中以字节为单位进行存储，取下一条指令时，应+4；对于 beq 指令，在 PC+4 之后，需要对 offet 左移两位。因此 32 位 PC 每次+4 后，取其 2 至 6 位作为地址输入 ROM 中。这种方式比较易于理解，符合 MIPS 的指令表达式

当使用 30 位 PC，在 ROM 中存储指令时，以字为单位进行存储，取下一条指令时，应+1。直接取 0 至 4 位作为地址输入 ROM 中。这种方式不需要将 offset 左移两位，比较简便。

2、现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

答：合理，无改进意见；

IM 为指令寄存器，以取指令为主，指令一旦写入便不再改变 于是使用 ROM 有一个优点是数据不能被改变，下次打开文件时指令仍然存在。而且，指令在程序运行期间不能改，因此 ROM 正符合这一特点。

DM 为数据寄存器，既存又取，需要能够随时写入与读出；于是使用 RAM，可读可写可复位，可以存放、修改数据，支持 sw 和 lw 指令。

GRF 为通用寄存器组，应该由 32 个寄存器组成，可存可取。是 CPU 的核心部件，Logisim 提供的寄存器功能与实际的 CPU 中相同，一切运算都必须通过寄存器完成。当然缺点也很明显，RAM 不支持跨地址读写数据，一次只能读写一个地址（4 字节），因此无法直接实现 lb，sb 等指令。

3、结合上文给出的样例真值表，给出 RegDst， ALUSrc， MemtoReg，RegWrite, nPC_Sel, ExtOp 与 op 和 func 有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

```
j=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&~opcode[0]

addu=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&~opcode[1]&&~opcode[0]&&funct[5]&&~funct[4]&&~funct[3]&&~funct[2]&&~funct[1]&&funct[0]

subu=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&~opcode[1]&&~opcode[0]&&funct[5]&&~funct[4]&&~funct[3]&&~funct[2]&&funct[1]&&funct[0]

ori=~opcode[5]&&~opcode[4]&&opcode[3]&&opcode[2]&&~opcode[1]&&opcode[0]
```

lw=opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&opcode[0]

sw=opcode[5]&&~opcode[4]&&opcode[3]&&~opcode[2]&&opcode[1]&&opcode[0]

beq=~opcode[5]&&~opcode[4]&&~opcode[3]&&opcode[2]&&~opcode[1]&&~opcode[0]

lui=~opcode[5]&&~opcode[4]&&opcode[3]&&opcode[2]&&opcode[1]&&opcode[0]

ExtOp[0]=ori||beq ExtOp[1]=lui||beq

MemtoReg=lw

MemWrite=sw

Branch=beq

ALUSrc=ori||lw||sw

RegDst=addu||subu

RegWrite=addu||subu||ori||lw||lui

nPC_Sel=branch&&zero（zero 是 ALU 的零信号）

4、充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

j=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&~opcode[0]

addu=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&~opcode[1]&&~opcode[0]&&funct[5]&&~funct[4]&&~funct[3]&&~funct[2]&&~funct[1]&&funct[0]

subu=~opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&~opcode[1]&&~opcode[0]&&funct[5]&&~funct[4]&&~funct[3]&&~funct[2]&&funct[1]&&funct[0]

ori=~opcode[5]&&~opcode[4]&&opcode[3]&&opcode[2]&&~opcode[1]&&opcode[0]

lw=opcode[5]&&~opcode[4]&&~opcode[3]&&~opcode[2]&&opcode[1]&&opcode[0]

sw=opcode[5]&&~opcode[4]&&opcode[3]&&~opcode[2]&&opcode[1]&&opcode[0]

beq=~opcode[5]&&~opcode[4]&&~opcode[3]&&opcode[2]&&~opcode[1]&&~opcode[0]

lui=~opcode[5]&&~opcode[4]&&opcode[3]&&opcode[2]&&opcode[1]&&opcode[0]

ExtOp[0]=ori||beq

ExtOp[1]=lui||beq

MemtoReg=lw

MemWrite=sw

Branch=beq

ALUSrc=ori||lw||sw

RegDst=addu||subu

RegWrite=addu||subu||ori||lw||lui

nPC_Sel=branch&&zero（zero 是 ALU 的零信号）

5、事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

不加入真值表，则到这条指令时，不能被识别为任何有效指令，等效于空指令。

6、前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM 片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

不需要修改，因为我们取的是地址的后 10 位（实际上是 5 位），无论起始地址设置为 0x0000 还是 0x3000 均被截断高位，起始地址自然会成为 0。对于偏移量不能跨地址的情况，可以把所有的 address 全部右移两位。这样，无需任何操作便可以正常运行。

如果超过地址限制，可以用多个存储器联合，2^n 个存储器需要 n 位片选信号，片选信号就等于地址的高位。然后用独热编码译码器接到写使能端上，用 MUX 接到读数据端即可。（实际的内存条分为多个片，也是通过片选信号连接）。

7、除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

对组合逻辑来说，不存在状态寄存器，其输出值 Z[t]不依赖于前面的输入值 X[t-i](1≤i≤t)。这时只要对每个输入向量证明其输出向量相同。

对一个时序电路而言，可以把它看成一个有限状态机(FSM，finite-state machine)。电路功能的等价可以用有限状态机的等价来判断。假定有两个状态机 A 和 B，要对它们进行比较。直观的说，当 A 和 B 有相同的接口，而且从相同的初始状态出发，两者对有效输入值序列产生相同的输出值序列，则可以说 A 和 B 等价。

形式验证的优点如下：

(1)形式验证是对指定描述的所有可能的情况进行验证，覆盖率达到了 100%。

(2)形式验证技术是借用数学上的方法将待验证电路和功能描述或参考设计直接进行比较，不需要开发测试激励。

(3)形式验证的验证时间短，可以很快发现和改正电路设计中的错误，可以缩短设计周期。

缺点是验证方法复杂抽象，难以准确把握。而且形式验证是数学逻辑分析，而不是电路分析，不能有效的验证电路的性能，如电路的时延和功耗等。